UNICAMP – Universidade Estadual de Campinas
FT – Faculdade de Tecnologia

Metodologias ágeis no contexto de desenvolvimento de software: XP, Scrum e Lean

Autores: Aline Cristine Fadel
Henrique da Mota Silveira

Limeira - 2010

Sumário

Resumo 4
Abstract 4
1. Introdução 5
2. Metodologias Ágeis 5
3. Estudo de Caso 20
4. Análise e Discussão sobre as Metodologias Ágeis Abordadas 22
5. Considerações Finais 24
6. Referências Bibliográficas 25

Resumo

Este trabalho apresenta um estudo sobre metodologias ágeis, mais especificadamente os métodos *Lean Software Development* (LSD), Scrum e *eXtreme Programming*. São apresentadas características, conceitos, formas de trabalho, além de uma comparação entre essas metodologias, como pontos em comum e diferenças. Este trabalho remete-se a disciplina FT027 (Gestão e Qualidade de Projetos) do programa de Pós Graduação da Faculdade de Tecnologia da Universidade Estadual de Campinas.

Abstract

This paper presents a study about agile methodologies, more specifically the methods Lean Software Development (LSD), Scrum and eXtreme Programming. Characteristics are presented, concepts, ways of working, beyond of comparison between these methodologies, as commonalities and differences. This work refers to FT027 (Management and Quality Project) of program by Graduate School of Technology, State University of Campinas.

1. Introdução

Na busca por lucros e eficiência, as empresas desenvolvedoras de *software* estão à procura por metodologias em que possam administrar melhor o seu tempo e seus recursos, para entregar produtos com qualidade.

Para que as empresas entreguem os produtos esperados pelos clientes e no tempo adequado, a metodologia Ágil surge como uma inovação, onde ele traz a eficiência para a equipe, pois o fluxo de desenvolvimento está extremamente organizado, desenvolvendo um *software* com o mínimo de recursos desperdiçados.

Este trabalho está organizado da seguinte maneira: a seção 2 apresenta três metodologias ágeis de desenvolvimento de software: *Lean Software Development* (LSD), Scrum e XP; a seção 3 descreve um estudo de caso do emprego da Metodologia Ágil; a seção 4 faz o comparativo entre as metodologias além de discutir a viabilidade de implantação de cada uma; e as considerações finais são apresentadas na seção 5.

2. Metodologias Ágeis

Nesta seção, será apresentada uma breve introdução sobre a origem dos métodos ágeis além de dar ênfase em três metodologias ágeis de desenvolvimento de software: *Lean Software Development* (LSD), Scrum e *eXtreme Programming* (XP).

2.1. Origem dos Métodos Ágeis

Filho [7, p. 22] sintetiza e coloca da seguinte maneira o modo pelo qual os métodos ágeis surgiram:

> *"Durante a evolução dos processos de Engenharia de Software, a indústria se baseou nos métodos tradicionais de desenvolvimento de software, que definiram por muitos anos os padrões para criação de software nos meios acadêmico e empresarial. Porém, percebendo que a indústria apresentava um grande número de casos de fracasso, alguns líderes experientes adotaram modos de trabalho que se opunham aos principais conceitos das metodologias tradicionais. Aos poucos, foram percebendo que suas formas de trabalho, apesar de não seguirem os padrões no mercado, eram bastante eficientes. Aplicando-as em vários*

projetos, elas foram aprimoradas e, em alguns casos, chegaram a se transformar em novas metodologias de desenvolvimento de software. Essas metodologias passaram a ser chamadas de leves por não utilizarem as formalidades que caracterizavam os processos tradicionais e por evitarem a burocracia imposta pela utilização excessiva de documentos. Com o tempo, algumas delas ganharam destaque nos ambientes empresarial e acadêmico, gerando grandes debates, principalmente relacionados à confiabilidade dos processos e à qualidade do software."

De acordo com Beck [3], em 2001 um encontro entre 17 líderes que trabalhavam no contra-fluxo dos padrões da indústria de software, foi discutido formas de trabalho, objetivando chegar a uma nova metodologia de produção de *software*, que pudesse ser usada por todos eles e em outras empresas, substituindo os modelos tradicionais de desenvolvimento. O grupo chegou ao consenso de que alguns princípios eram determinantes para a obtenção de bons resultados. O resultado deste encontro foi a identificação de 12 princípios e a publicação do Manifesto Ágil [3] que os representa com quatro premissas:

- Indivíduos e iterações são mais importantes do que processos e ferramentas;
- Software funcionando é mais importante do que documentação completa;
- Colaboração com o cliente é mais importante do que negociação de contratos;
- Adaptação a mudanças é mais importante do que seguir o plano inicial;

Kalermo e Rissanen [10] apontam os 12 princípios mencionados acima:

1. Prioridade é satisfazer o cliente através da entrega contínua e adiantada de software com valor agregado;
2. As mudanças nos requisitos são bem-vindas, mesmo tardiamente no desenvolvimento. Processos ágeis tiram vantagem das mudanças visando vantagem competitiva para o cliente;
3. Frequentes entregas do *software* funcionando, de poucas semanas a poucos meses, com preferência à menor escala de tempo;
4. As pessoas de negócio e desenvolvedores devem trabalhar diariamente em conjunto por todo o projeto;

5. Os projetos devem ser construídos em torno de indivíduos motivados. Dando o ambiente e o suporte necessário e confiança para fazer o trabalho;
6. O método mais eficiente e eficaz de transmitir informações para e entre uma equipe de desenvolvimento é através de conversa face a face;
7. *Software* funcionando é a medida primária de progresso;
8. Os processos ágeis promovem desenvolvimento sustentável. Os patrocinadores, desenvolvedores e usuários devem ser capazes de manter um ritmo constante indefinidamente;
9. Contínua atenção a excelência técnica e bom design aumentam a agilidade;
10. Simplicidade para maximizar, a quantidade de trabalho não realizado é essencial;
11. As melhores arquiteturas, requisitos e designs emergem de equipes auto-organizáveis;
12. Em intervalos regulares, a equipe deve refletir sobre como tornar-se mais efetiva, e então, ajustar-se de acordo com seu comportamento.

O Manifesto Ágil, segundo Filho [7], ressalta o que mais tem valor para as metodologias ágeis, a importância de como saber lidar com pessoas, assim como ter o cliente colaborando para encontrar a melhor solução, entregar o *software* com qualidade e do que se adaptar às mudanças. Isto oculta parte das dificuldades dos processos, contratos, documentação e planejamento para o desenvolvimento de um *software* [7].

2.2. *Lean Software Development* (LSD)

O sistema Toyota de Produção [12], também conhecido como *Lean Manufacturing*, surgiu no Japão, na fábrica de automóveis Toyota, logo após a Segunda Guerra Mundial. Nesta época a indústria japonesa tinha uma produtividade muito baixa e uma enorme falta de recursos, o que naturalmente a impedia adotar o modelo de produção em massa.

De acordo com o *Lean Institute* Brasil [11], "*Lean* é uma estratégia de negócios para aumentar a satisfação dos clientes através da melhor utilização dos recursos. A Gestão *Lean* procura fornecer consistentemente valor aos clientes com os custos mais baixos (Propósito) através da identificação de melhoria dos fluxos de valor primários e de suporte (Processos) por meio do envolvimento das pessoas qualificadas, motivadas e com iniciativa (Pessoas). O foco da implementação deve estar nas reais necessidades dos negócios e não na simples aplicação das ferramentas *lean*."

De acordo com Mary Poppendieck [14], o desenvolvimento de *software Lean* é a aplicação dos princípios da Toyota *product development system* para o desenvolvimento de *software*. Quando ele é aplicado corretamente, o desenvolvimento é de alta qualidade, além de ser realizado rapidamente e possuir um baixo custo. Além disso, o desenvolvimento ágil possui um grande sucesso devido ao *Lean*.

2.2.1 Princípios

O pensamento *Lean* foca em oferecer o que o cliente quer, onde e quando quiserem sem haver qualquer desperdício. Para atingir esse objetivo, *Lean* possui alguns princípios. No trabalho publicado por Poppendieck Poppendieck [15], é feito o mapeamento dos sete princípios de *Lean* para o desenvolvimento de software. Em conjunto todos esses princípios oferecem entregas rápidas, com mais qualidades e com baixo custo, ao mesmo tempo. A seguir cada princípio é apresentado conforme o ponto de vista de Filho [7].

2.2.1.1 Eliminar o desperdício

O desperdício em si pode acontecer sob vários pontos de vista, dentre eles, desperdício de: dinheiro, recursos, tempo, esforço e espaço. Cada etapa e atividade realizada no processo deve necessariamente contribuir para que o produto final seja construído mais rapidamente, com mais qualidade ou a um custo mais baixo. Filho [7] coloca uma série de cenários, a seguir, onde o desperdício é evidente.

A existência de **funcionalidades incompletas** gera desperdício porque despendem esforços para serem iniciadas e não adicionam valor ao *software*.

Pedaços de **código incompletos** tendem a se tornar obsoletos, mais difíceis de serem integrados e os programadores lembram menos a respeito da intenção inicial do código.

Excesso de processos é um desperdício porque eles demandam recursos e aumentam o tempo para a conclusão das tarefas.

A **criação de documentos** infla o processo e causa desperdício, pois eles consomem tempo para serem produzidos, sem garantias de que alguém irá lê-los. Documentos ficam desatualizados e podem ser perdidos, tornam a comunicação mais lenta e reduzem o poder comunicativo, pois é um meio de comunicação de via única no qual não é possível que escritor e leitor interajam em tempo real. Além disso, muitas vezes, documentos representam apenas formalismos burocráticos que não acrescentam valor ao *software*.

Processos complexos aumentam a quantidade de documentos, por isso também caracterizam desperdício.

Antecipar funcionalidades é um desperdício porque aumenta a complexidade do *software* desnecessariamente com mais código, mais esforços com testes e mais integrações.

Troca de tarefas é uma forma de desperdício porque um número excessivo de mudanças de contexto reduz a produtividade. Alocar desenvolvedores em mais de um projeto é um desperdício porque as necessidades de um projeto não levam em conta a situação dos outros.

Esperas por requisitos, por testes, por aprovação ou por *feedback* retardam o fluxo do desenvolvimento ou a identificação de problemas.

Defeitos são desperdício porque o custo para corrigi-los aumenta com o tempo. À medida que o projeto evolui, a complexidade do código aumenta, com isso, a localização e a remoção de um defeito torna-se mais difícil

Portando, processos que envolvam comunicação e atividades de gerenciamento devem ser o mais simples e objetivo possível, para que menos pessoas sejam necessárias, menos etapas sejam cumpridas até a conclusão de um ciclo e, assim, o processo inteiro será mais rápido e menos custoso [7].

2.2.1.2 Amplificar o aprendizado

Filho [7] afirma que lições devem ser extraídas das experiências vividas pela equipe e incorporadas ao processo, fazendo com que as dificuldades passadas sejam fonte de conhecimento e contribuam para o amadurecimento da equipe e do processo. O uso de um processo definido engessa o aprendizado.

Deming [5 *apud* [7, p. 78]] propôs um ciclo de melhoria contínua que pode ser aplicado neste cenário, sendo que primeiro deve-se fazer a identificação o problema, localizar a sua causa, criar uma solução, implementar, verificar os resultados e se adaptar à nova realidade.

2.2.1.3 Adiar comprometimentos e manter a flexibilidade

Adiar decisões permite que as escolhas sejam apoiadas por mais experiência e conhecimento adquiridos no decorrer do processo. Para retardar decisões durante a construção de sistemas é importante que a equipe crie a capacidade de absorver mudanças tratando os planejamentos como estratégias para atingir um objetivo e não

como comprometimentos. Assim, mudanças serão vistas como oportunidades para aprender e atingir as metas [7].

2.2.1.4 Entregar rápido

De acordo com Filho [7], com ciclos rápidos o desenvolvimento caminha através de um processo iterativo no qual o cliente refina suas necessidades e as obtém implementadas já no próximo ciclo. Iterações curtas trazem mais experiência para a equipe e aumentam a sua segurança para tomar decisões.

Até recentemente, a rápida entrega de *software* não era valorizada, a estratégia de não cometer erros era vista como mais importante. Porém, o desenvolvimento rápido do *software* tem diversas vantagens, já que a velocidade de desenvolvimento também ajuda atender às necessidades atuais do cliente, além de permitir adiar a tomada de decisões para quando for acumulado conhecimento suficiente para tal [8].

2.2.1.5 Tornar a Equipe Responsável

Sendo que a equipe de desenvolvimento é responsável pela confecção do produto que é entregue ou usado pelo cliente, o *software*, logo o conhecimento dos detalhes técnicos deve ser levado em consideração na tomada de decisões a na definição de processos [7].

Franco [8] coloca que envolver os desenvolvedores nas decisões de detalhes técnicos é fundamental para atingir a excelência. Quando dotados com a experiência necessária e guiados por um líder, eles tomarão decisões técnicas e de processos, melhores que qualquer outra pessoa poderia tomar por eles. *Lean* utiliza as técnicas de produção puxada (*pull*) para agendar o trabalho e possuem mecanismos de sinalizações locais, de forma a permitir que outros trabalhadores identifiquem o trabalho que necessita ser realizado. No desenvolvimento de software *Lean*, o mecanismo da produção puxada (*pull*) corresponde ao acordo de entregar versões refinadas e incrementais do software em intervalos regulares. A sinalização local é feita através de gráficos visuais, reuniões diárias, integrações freqüentes e testes automatizados [8].

Franco [8] apresenta dois exemplos de representações gráficas (Figuras 1 e 2). A Figura 1 ilustra um quadro que pode ser utilizado para distribuir as funcionalidades a serem realizadas em cada iteração, semelhante ao *kanban* do da metodologia *Lean*. É utilizado para nivelar e controlar o fluxo de produção, definindo a quantidade de trabalho a ser realizado em cada iteração. As colunas representam a segmentação do trabalho a ser realizado através das iterações. Em cada coluna é acrescentado um cartão

que corresponde ao objetivo da iteração (Cartão Tema), e abaixo deles são colocados cartões correspondentes aos requisitos a serem implementados. Estes cartões podem ser relacionados à arquitetura ou às funcionalidades do produto. O nivelamento da produção é feito através da quantidade de trabalho a ser realizado para transformar os requisitos descritos nos cartões em incrementos do produto; o esforço alocado para cada iteração deve ser homogêneo, não podendo haver grandes variações entre elas.
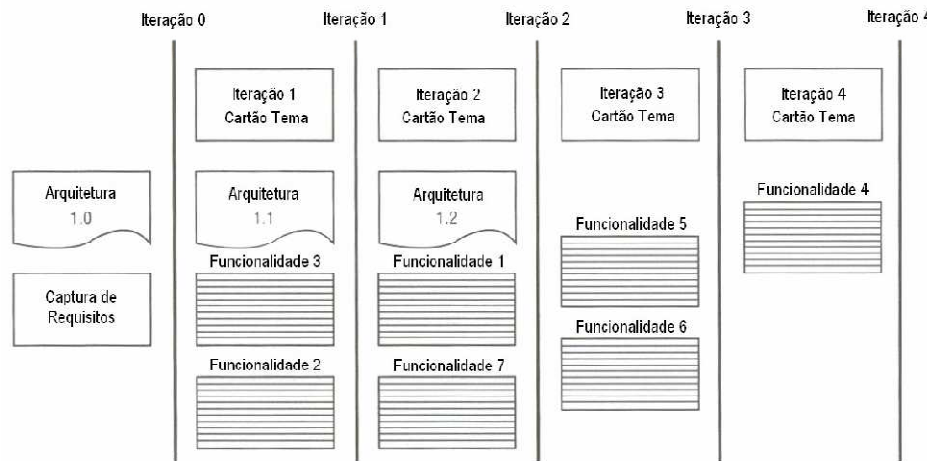
Figura 1. Quadro de cartões de funcionalidades [15 *apud* [8, p. 49]].

A Figura 2 ilustra um exemplo dos controles visuais utilizados para acompanhar a execução do projeto. A curva azul representa a estimativa de esforço necessário para finalizar as tarefas remanescentes na iteração atual e a linha amarela corresponde à quantidade de funcionalidades remanescentes para serem codificadas e incorporadas ao incremento do produto a ser entregue no fim da iteração.
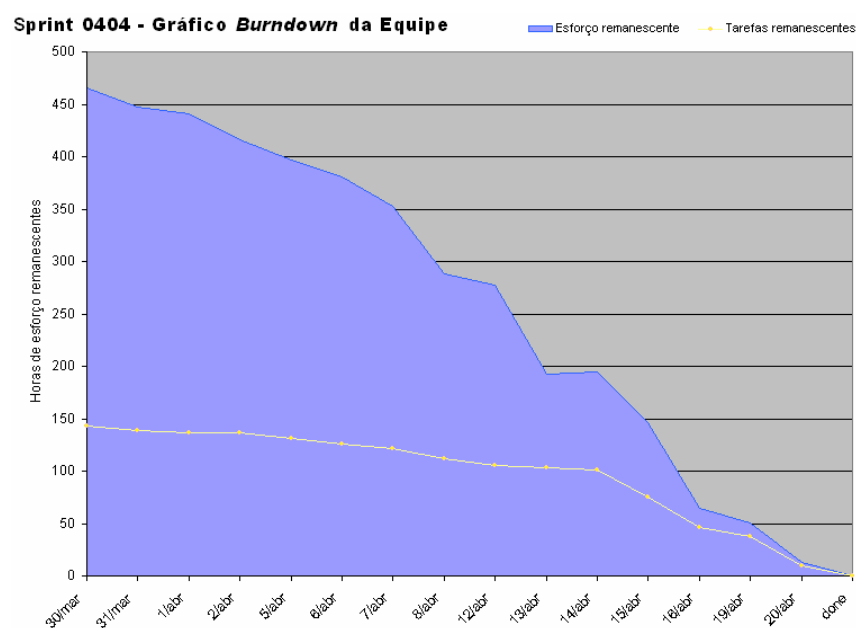
Figura 2. Exemplo de gráfico do tipo *burndown* [8, p. 50].

2.2.1.6 Construir Integridade

De acordo com Filho [7], a equipe de desenvolvimento deve elaborar soluções que a deixem segura de que estão construindo um produto de qualidade. Por isso existe a necessidade da utilização de uma arquitetura adequada, mantendo uma alta cobertura de testes automatizados e preservando a flexibilidade para mudanças, adaptações e extensões são formas de trazer segurança e motivação para alcançar níveis mais elevados de qualidade. Dessa maneira, ao invés de se gastar tempo para encontrar e corrigir defeitos, mais investimento e esforços devem ocorrer na prevenção através de vários tipos de teste. De acordo com Franco [8], o *software* necessita de um nível adicional de integridade e precisa manter sua utilidade ao longo do tempo. O *software* que possui integridade possui uma arquitetura coerente, facilidade satisfatória de uso, atende aos propósitos para o qual foi proposto, manutenível, adaptável e extensível.

2.2.1.7 Visualizar o Todo

Para Franco [8], para que haja a obtenção da integridade nos sistemas de grande complexidade é preciso um conhecimento profundo de diversas áreas. Filho [7] coloca que a criação de grandes sistemas envolve soluções integradas que devem prover bons resultados perante uma análise global do *software*. Os pontos de vista dos clientes e dos usuários equivalem a visões de alto nível do sistema. Otimizações macro canalizam os esforços para aumentar a satisfação dos usuários finais através de um produto consistente. *Lean* ainda recomenda a escolha de métricas de alto nível que sejam representativas para identificar a evolução. Essas métricas devem levar em consideração também a qualidade e a satisfação do cliente, pois a partir delas será possível avaliar quais trocas são vantajosas.

2.3. Scrum

Inicialmente o Scrum foi concebido para gerenciamento de projetos de fabricação de automóveis e de produtos de consumo, por Takeuchi e Nonaka no artigo “*The new product development game*”, em janeiro-fevereiro de 1986, pela Universidade de *Harvard*. Neste artigo, Takeuchi e Nokada notaram que em projetos com equipes pequenas e multidisciplinares produziam melhores resultados, e associaram isto a formação Scrum do Rugby. Em 1995, o Scrum teve sua definição formalizada por Ken Schwaber, que trabalhou para consolidá-lo como método de desenvolvimento de software por todo o mundo [1] [16].

O Scrum não define uma técnica específica para o desenvolvimento de software durante a etapa de implementação, ele se concentra em descrever como os membros da equipe devem trabalhar para produzir um sistema flexível, num ambiente de mudanças constantes. A idéia central do Scrum é que o desenvolvimento de sistemas envolve diversas variáveis (ambientais e técnicas) e elas possuem grande probabilidade de mudar durante a execução do projeto (por exemplo: requisitos, prazos, recursos, tecnologias etc.). Estas características tornam o desenvolvimento do sistema de *software* uma tarefa complexa e imprevisível, necessitando de um processo flexível e capaz de responder às mudanças [8].

2.3.1.1. Elementos de Apoio

Filho [7] descreve os elementos de apoio do Scrum, sendo que os únicos elementos que a equipe produz para seguir a práticas de Scrum são cartões com as funcionalidades e gráficos de acompanhamento. Os cartões agrupados formam o *Backlog* do Produto e outros *backlogs*. Os gráficos são atualizados freqüentemente e devem refletir o estado do projeto. Estes são listados a seguir [7]:

- ***Backlog* do Produto:** Lista de todos os cartões de funcionalidades que o produto deve possuir e que ainda não foram desenvolvidas;
- ***Backlog* Selecionado:** Um subconjunto de funcionalidades que o cliente escolheu a partir do *backlog* do produto para ser implementado no *sprint* atual e que não pode ser modificado durante o *sprint*;
- ***Backlog* do *Sprint*:** Lista priorizada, obtida a partir da quebra dos cartões do *backlog* selecionado em tarefas menores;
- ***Backlog* de Impedimentos:** Lista dos obstáculos identificados pela equipe que não pertencem ao contexto do desenvolvimento;
- **Gráficos de Acompanhamento:** Gráficos que medem a quantidade de trabalho restante (*burndown charts*) são os preferidos em Scrum. É recomendado fazê-los para várias esferas do projeto: para o produto, para a *release* e para o *sprint*.

2.3.1.2. Papéis e Responsabilidades

De acordo com Franco [8], os papéis no Scrum que possuem tarefas e propósitos diferentes durante o processo e suas práticas são apresentados a seguir:

- **Cliente:** participa das tarefas relacionadas à definição da lista de funcionalidade do software sendo desenvolvido ou melhorado, elaborando os requisitos e restrições do produto final desejado.
- **Gerente:** é encarregado pela tomada das decisões finais, utilizando as informações visuais disponibilizadas graficamente pelos padrões e convenções a serem seguidas no projeto. Ele também é responsável por acordar, junto aos Clientes, os objetivos e requisitos do projeto.
- **Equipe Scrum:** é a equipe de projeto que possui autoridade de decidir sobre as ações necessárias e de se organizar para poder atingir os objetivos preestabelecidos. A Equipe Scrum é envolvida, por exemplo, na estimativa de esforço, na criação e revisão da lista de funcionalidade do produto, sugerindo obstáculos que precisam ser removidos do projeto.
- **Scrum Master:** é responsável por garantir que o projeto esteja sendo conduzido de acordo com as práticas, valores e regras definidas no Scrum e que o progresso do projeto está de acordo com o desejado pelos Clientes. O Scrum Master interage tanto com a Equipe Scrum, como com os Clientes e o Gerente durante o projeto. Ele também é responsável por remover e alterar qualquer obstáculo ao longo do projeto, para garantir que a equipe trabalhe da forma mais produtiva possível.
- **Responsável pelo Produto:** é oficialmente responsável pelo projeto, gerenciamento, controle e por tornar visível a lista de funcionalidade do produto. Ele é selecionado pelo Scrum Master, Clientes e Gerente. Ele também é responsável por tomar as decisões finais referentes às tarefas necessárias para transformar a lista de funcionalidades no produto final, participando na estimativa do esforço de desenvolvimento necessário e é responsável pelo detalhamento das informações referentes à lista de funcionalidade utilizada pela Equipe Scrum.

2.3.1.3. Entregas Contínuas

A metodologia proposta pelo modelo Scrum aplica um sistema de entregas contínuas. Nesta metodologia com os *backlogs* definidos *(*que são os requisitos funcionais do sistema*)*, um *sprint* programado (tempo predeterminado no qual será dividido o trabalho para efetuação de uma entrega, tendo como padrão o prazo dentre duas a quatro semanas), reuniões diárias (de 10 minutos para acompanhar se o projeto

está de acordo com o planejamento) e, ao final de cada *sprint*, uma reunião de retrospectiva e planejamento do próximo *sprint* (Figura 3) [1].
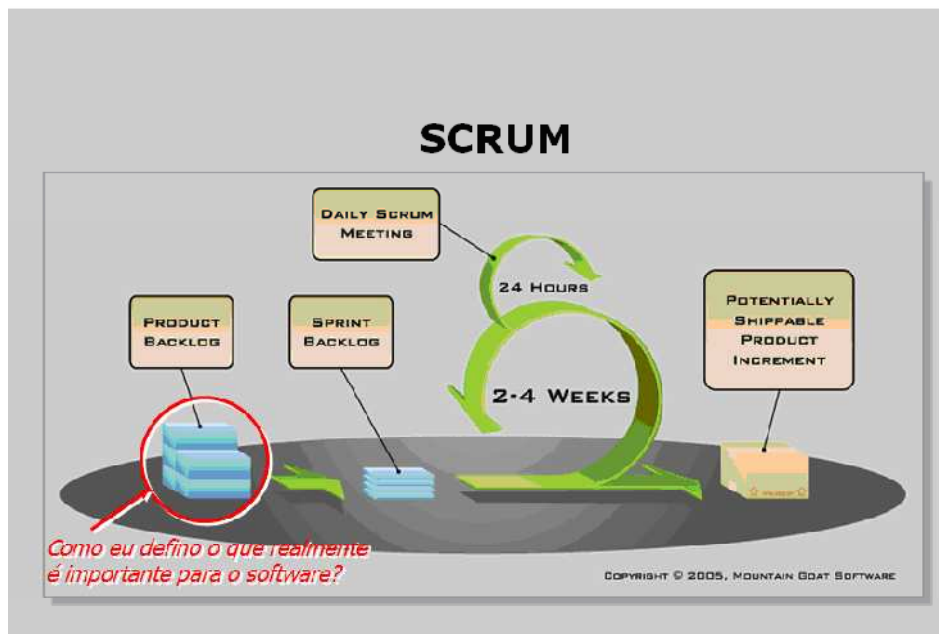
Figura 3. Ciclo de desenvolvimento Scrum [1, p. 3].

2.3.1.4. Planejando as interações (entregas)

Na metodologia Scrum para se planejar uma interação (uma entrega) deve-se seguir a seguinte seqüência de atividades, segundo Araujo e Galina [1]:

- Discutir uma estória;
- Decompor uma estória em tarefas;
- Desenvolvedor deve aceitar a responsabilidade por cada tarefa;
- Após todas as estórias terem sido discutidas e todas as tarefas aceitas, os desenvolvedores individualmente estimam as tarefas aceitas para ter certeza de que eles não estão se comprometendo com algo que não podem cumprir.

2.3.1.5. Desempenho da equipe e Escopo de Utilização

O desempenho da equipe será definido com base no histórico de entregas e conclusões de projetos anteriores, ações que iram definir o quanto em pontos a equipe consegue atingir. Através de um sistema de pontuação cada desenvolvedor atinge em média 20 pontos por *sprint* de duração de duas semanas. Tal desempenho pode ser afetado por diversos fatores, os quais que devem ser trazidos para discussão na reunião de retrospectiva e planejamento. Nesta reunião, deve-se abordar a apresentação de problemas e dificuldade encontrados, buscando assim resoluções, melhorias no processo

de desenvolvimento de software e conseqüentemente objetivando a velocidade de conclusão dos próximos *sprints* [1].

O Scrum é uma metodologia destinada a pequenas equipes com menos de dez pessoas. Schwaber e Beedle [16 apud [8]] sugerem que a equipe seja composta de cinco a nove integrantes, se mais pessoas estiverem envolvidas no projeto, devem-se formar múltiplas Equipes Scrum.

2.4. *Extreme Programming* (XP)

Segundo Franco [8], a metodologia *Extreme Programming* (XP) surgiu como uma tentativa para solucionar os problemas causados pelos ciclos de desenvolvimento longos dos modelos de desenvolvimento tradicionais. O XP é composto por práticas que se mostraram eficientes nos processos de desenvolvimento de *software* nas últimas décadas. Depois de aplicado e obtido sucesso num caso real, o XP foi formalizado através de princípios chaves e doze práticas [2]. Quando analisadas individualmente, as práticas que compõem o XP não são novas. No XP estas práticas foram reunidas e alinhadas de tal forma a criar uma interdependência entre elas, e assim, deu origem a uma nova metodologia de desenvolvimento de software.

De acordo com Franco [8], os quatro princípios chaves do XP são a **Comunicação**, **Simplicidade**, ***Feedback*** e a **Coragem**. Filho [7] complementa estes princípios com mais um item: **Respeito**. A seguir estes são detalhados:

- **Comunicação:** muitos dos problemas que ocorrem no decorrer do projeto podem ser relacionados com problemas de comunicação entre a equipe ou entre a equipe do projeto e o próprio cliente. Uma pessoa pode deixar de comunicar um fato importante à outra pessoa, um programador pode deixar de levantar uma questão importante ao cliente etc. O XP mantém o fluxo de comunicação através de algumas práticas que não podem ser realizadas sem comunicação. Exemplos disto são: testes de unidade, programação em pares e estimativa do esforço de cada tarefa;
- **Simplicidade:** deve-se sempre selecionar a alternativa mais simples que possa funcionar. O XP se baseia no fato que é mais barato fazer algo mais simples e alterá-lo conforme as necessidades forem surgindo do que tentar prever as necessidades futuras, introduzindo uma complexidade que possa vir a não ser necessária no futuro;

- ***Feedback*:** todo problema deve ser evidenciado o mais cedo possível para que possa ser corrigido imediatamente. Toda a oportunidade é descoberta o mais cedo possível para que possa ser incorporada de forma rápida ao produto que está sendo construído;
- **Coragem:** é preciso coragem para apontar um problema no projeto, para solicitar ajuda quando necessário, para simplificar o código que já está funcionando (podendo introduzir novos defeitos), comunicar ao cliente que não será possível implementar um requisito no prazo estimado e, até mesmo, para fazer alterações no processo de desenvolvimento.
- **Respeito**: é necessário entre as pessoas, sendo a peça principal para que os demais valores funcionem. Sem respeito, a Comunicação e o *Feedback* serão pouco eficientes e a Coragem de um membro poderá ser nociva aos demais por não estar alinhada com os interesses da equipe. Todos os participantes devem manter o respeito entre si e em relação aos seus trabalhos. Desvalorizar alguém, a função que exerce ou a qualidade de seu trabalho são formas de falta de respeito que minam a sinergia da equipe. Uma forma de respeito de cada um para com a equipe é assumir responsabilidades que serão capazes de cumprir e da equipe para com seus membros é confiar que cada um fará o melhor trabalho possível. XP considera que os desenvolvedores também são pessoas e preocupa-se com suas necessidades pessoais e profissionais através dos princípios da humanidade e da diversidade, estabelecendo compromissos com o princípio da aceitação da responsabilidade.

2.4.1. Papéis e Responsabilidades

Existem diferentes papéis sugeridos pela metodologia XP para diferentes fases, práticas e ferramentas necessárias ao longo do projeto. A seguir, de acordo com Beck [2], estes papéis são descritos:

- **Programador:** escrevem testes e mantém o programa o mais simples e conciso possível. A primeira característica que torna o XP possível é a habilidade de comunicação e coordenação com outros membros da equipe;
- **Cliente:** escreve as estórias e os testes funcionais, além de decidir quando cada requisito foi satisfeito. O cliente também define a prioridade de implementação de cada requisito;

- **Testador:** ajuda o cliente a escrever os testes funcionais. Ele também realiza os testes funcionais regularmente, comunicando os resultados dos testes e mantém o conjunto de testes;
- **Monitor:** fornece a realimentação para a equipe do projeto. Ele acompanha a conformidade das estimativas feitas pela equipe de desenvolvimento (por exemplo, estimativas de esforço) e fornece comentários de quanto acuradas elas estão, para poder melhorar futuras estimativas. Ele também acompanha o progresso de cada iteração e avalia se o objetivo é viável dentro das limitações de tempo e recursos, ou se alguma mudança é necessária no processo;
- **Treinador:** é a pessoa responsável pelo processo como um todo. Um profundo conhecimento do XP é importante para este papel, pois é ele que guiará os outros envolvidos no projeto a executar o processo de forma adequada;
- **Consultor:** é um membro externo com conhecimento técnico específico necessário para o projeto. O consultor auxilia a equipe a resolver problemas específicos;
- **Chefe:** responsável pelas tomadas de decisões. Para isso, ele comunica-se com a equipe de projeto para determinar a situação atual e para identificar qualquer dificuldade ou deficiência do processo.

2.4.2. Práticas

Assim como papéis e responsabilidades, existe na metodologia XP um conjunto de práticas formado por doze itens, que de acordo com Franco [8], são descritos a seguir:

- **Jogo de planejamento:** nesta prática existe uma grande interação entre o Cliente e os Programadores. Os Programadores estimam o esforço necessário para implementar as estórias definidas pelo Cliente e este, decide sobre o escopo e duração das iterações;
- **Incrementos curtos e pequenos:** um incremento simples e funcional é gerado rapidamente pelo menos uma vez a cada dois ou três meses. Desta forma é possível ter um retorno por parte do Cliente em tempo hábil para poder incorporar mudanças e corrigir o produto sendo desenvolvido;
- **Metáfora:** é elaborado uma descrição que permite todos envolvidos no projeto (Clientes, Programadores, Gerente etc.) explicar como o sistema funciona. Ela cria uma visão comum e sugere uma estrutura de como o

problema e a solução são percebidos no contexto do sistema sendo produzido. Ela também auxilia os envolvidos a compreender os elementos básicos do sistema e seus relacionamentos, criando um vocabulário comum para o projeto;

- **Projeto simples:** o sistema deve ser projetado da forma mais simples possível de acordo com as necessidades atuais do projeto. As complexidades desnecessárias são removidas assim que são descobertas;
- **Testes:** o desenvolvimento do *software* é dirigido por testes. Os testes de unidade são desenvolvidos antes da codificação e são executados continuamente. Os testes funcionais (aceitação) são escritos pelo Cliente;
- **Reestruturação:** melhoria do sistema através da remoção de duplicações de código, melhorando a comunicação, simplificando e adicionando flexibilidade;
- **Programação em pares:** dois programadores escrevem o código em um único computador;
- **Propriedade coletiva:** qualquer programador pode alterar qualquer parte do código em qualquer lugar do sistema a qualquer momento;
- **Integração contínua:** o sistema é integrado e são geradas versões internas, diversas vezes ao dia, sempre que uma estória é finalizada;
- **Semanas de 40 horas:** não se devem trabalhar mais do que quarenta horas por semana, isto deve ser encarado como uma regra;
- **Cliente no local:** um usuário real do produto (Cliente) deve ser adicionado à equipe de programadores. Ele deve estar disponível em tempo integral para responder as eventuais dúvidas;
- **Padrão de codificação:** os programadores escrevem todo o código de acordo com regras que enfatizam a comunicação durante a codificação. Antes do início do projeto deve ser definido um padrão que deverá ser seguido por toda a equipe de Programadores.

As práticas do XP, segundo Franco [8], são agrupadas em quatro grupos de acordo com sua finalidade, começando da elipse central para a extremidade:

- Codificação;
- Equipe;
- Processo;
- e Produto.

A Figura 4 apresenta uma ilustração gráfica do agrupamento das práticas do XP, descritas anteriormente.
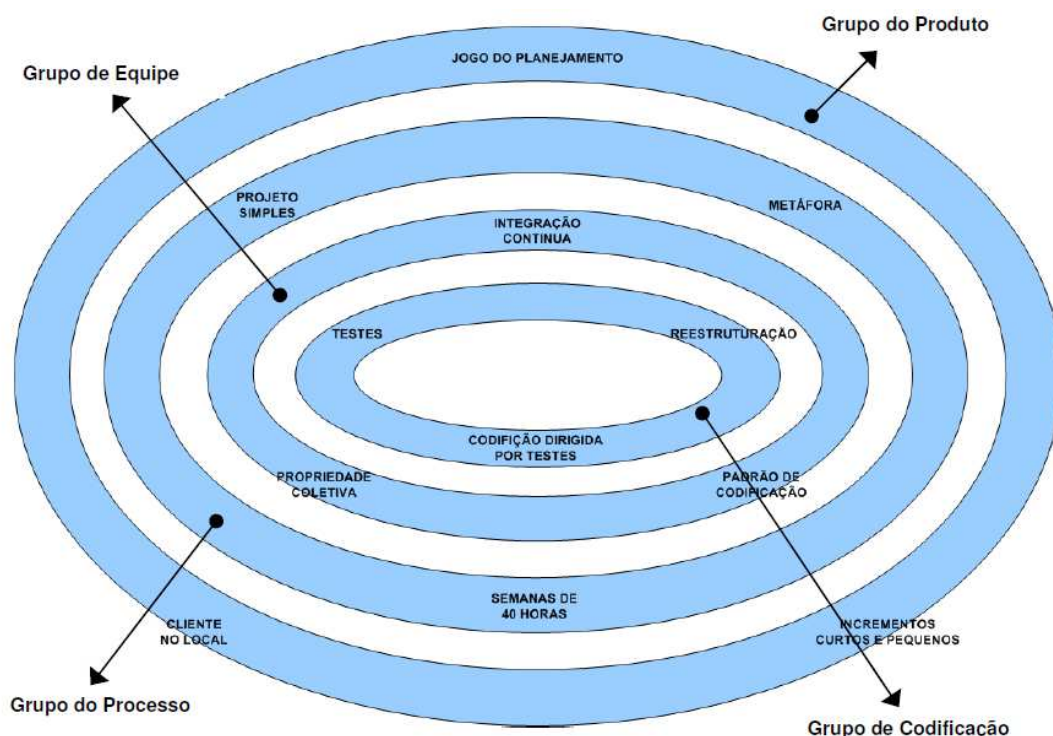
Figura 4. Agrupamento das práticas do XP [8, p. 32].

3. Estudo de Caso

O estudo de caso foi retirado de "*Introducing Lean Principles with Agile Practices at a Fortune 500 Company*" e foi elaborado por Emma Parnell-Klabo (2006).

A *Capital One* é uma grande empresa do setor financeiro e está no *ranking* da *Fortune*. Porém ela necessitava diminuir os seus custos e aumentar a competividade no mercado, baseado nisso ela optou pelo *Lean* Ágil no desenvolvimento de seus produtos de *software*.

Antes de qualquer ação ela necessitou reorganizar seus processos, e para isso utilizou os princípios DMAIC do Seis Sigma.

Utilizando os princípios do *Lean*, foram identificados pontos fortes de desperdícios: transporte, inventário, espera, processamento, e os defeitos (retrabalho).

Os fatores que deveriam ser mais trabalhados eram:

- Aperfeiçoamento do tempo de recurso com atribuição de três ou mais projetos ao mesmo tempo;
- Execução manual dos casos de testes sem os benefícios da automação;

- Paradas freqüentes no processo de desenvolvimento que acarretavam atrasos de alguns dias e até semanas nos produtos;
- Os executantes do processo possuíam pouca visibilidade de todas as outras tarefas;
- Redundância e sobreposição de papéis entre os departamentos;
- Expectativas pouco claras dos clientes;
- Baixo nível de interação entre os clientes e o time de desenvolvedores;
- Atribuir os recursos mais qualificados e conhecedores técnicos para coordenar o trabalho em vez de utilizá-los para contribuir no processo de desenvolvimento;

Uma análise no projeto verificou que o código (projeto e desenvolvimento) representa apenas 10 % de todo o projeto. Ou seja, havia muito desperdício em outras tarefas.

Após a aplicação do desenvolvimento ágil Scrum, o time de desenvolvimento diminuiu em 50% o número de tarefas.

Na escolha de um projeto piloto, o time focou em dois pontos:

- O projeto precisaria ser similar em tamanho e complexidade de outros projetos já entregues;
- Os membros do time necessitariam possuir diferentes conhecimentos e habilidades.

Os resultados desse piloto foram: 30% menos na codificação e testes, e 15% menos com os custos com recursos, além da duração do projeto ser 40% menor.

Apesar de todos esses ganhos, o time de desenvolvimento enfrentou algumas dificuldades, como a falta de apoio do executivo.

Diante desses resultados foi observado que a junção de *Lean* e Ágil forneceram bons resultados, porém, antes da aplicação dos princípios do *Lean* e técnicas de software Ágil, as ferramentas *Lean* devem ser utilizadas para remover resíduos, após todos os processos estarem organizados.

4. Análise e Discussão sobre as Metodologias Ágeis Abordadas

As metodologias ágeis utilizadas em desenvolvimento de *software* quebram o paradigma do desenvolvimento em cascata, e outros processos mais tradicionais, porém elas não são uma substituição aos processos já existentes, mas sim uma complementação ou uma alternativa.

A cada nova iteração um subprojeto é elaborado, onde são realizadas todas as etapas como: planejamento, requisitos, codificação e testes. E essas iterações duram poucas semanas, o que leva a resultados rápidos, principalmente para os clientes. Os subprojetos entregues frequentemente possuem funcionalidades já em operação.

Os grupos de desenvolvimento ágeis geralmente são pequenos, facilitando a comunicação, a qual normalmente ocorre em tempo real, o que resulta em pouca documentação do projeto, sendo isso um ponto desfavorável.

O cliente é parte da equipe de desenvolvimento, realiza sugestões e melhorias, participa do planejamento do escopo de cada iteração, e aprova cada entrega. O projeto tem grandes ganhos com a participação do cliente, pois as mudanças podem ocorrer no início de cada fase sem comprometer a qualidade e os custos do desenvolvimento.

O desenvolvimento Scrum foi criado inicialmente para o gerenciamento de projetos. Já o XP é utilizado no desenvolvimento, devido à aplicação de suas técnicas e ferramentas. O *Lean* não é uma prática de desenvolvimento ou um gerenciamento de projetos, mas sim um conjunto de princípios, valores e ferramentas que tornam o desenvolvimento enxuto. Com essas observações, pode-se afirmar que XP, Scrum e *Lean* podem ser utilizados como metodologias complementares.

Em suma, Franco [8] aponta as características comuns entre as metodologias ágeis da seguinte forma:

- **Testes:** Métodos tradicionais tratam a implementação e os testes como fases completamente distintas. Em métodos ágeis, esta segmentação tende a desaparecer. Implementação e testes acontecem muitas vezes juntos, onde o mesmo programador que cria o código, também o testa;
- **Desenvolvimento Iterativo:** O desenvolvimento acontece em ciclos (iterações), que têm o objetivo de produzir e integrar partes do software. Cada iteração pode durar desde alguns meses até poucas horas, conforme a metodologia escolhida e as habilidades da equipe. Desta forma, o processo

torna-se flexível para acomodar mudanças funcionais e de prioridade durante a construção do sistema. No fim de cada ciclo, o software pode ser entregue ao cliente para que o restante do desenvolvimento seja direcionado pelo seu *feedback*.

- **Desenvolvimento Incremental:** Durante as iterações, o software pode receber incrementos funcionais de duas formas: 1) acrescentando funcionalidades à medida que o software cresce, ou 2) evoluindo as funcionalidades junto com o sistema. Na primeira, as funcionalidades são implementadas completamente e entregues uma por vez, no segundo caso elas são criadas de forma simplificada para entrarem em produção rapidamente e, se preciso, são completadas ou melhoradas nas próximas iterações;
- **Colaboração:** Clientes e usuários estão mais próximos dos desenvolvedores e acompanham a evolução do produto. O contato constante com o cliente permite *feedback* rápido e facilita a comunicação. Com mais comunicação, a visão dos participantes sobre o andamento do projeto torna-se mais apurada, evitando que surpresas aconteçam no fim do projeto. A identificação e resolução de problemas tornam-se mais rápidas e as prioridades, o escopo e os detalhes da implementação podem ser negociados com mais facilidade;
- **Estimativas:** Métodos ágeis usam estimativas ao invés de predições. Estimativas são compostas por uma dupla de valores $< v, p >$, onde v é um palpite sobre determinado evento ou atividade e p é a probabilidade de v acontecer. Equipes ágeis baseiam-se na comunicação e na transparência. Ao invés de tratar suas estimativas como fatos, admitem que existe uma incerteza associada ao valor estimado e evidenciam isso para que o cliente e outros envolvidos também tomem ciência do grau de dificuldade de cada tarefa. Estimativas de longo prazo geralmente possuem graus maiores de incerteza associados. À medida que o tempo passa e o conhecimento sobre o assunto aumenta, as estimativas podem ser refeitas considerando um nível maior de detalhes e associando probabilidades de sucesso mais altas;
- **Negociação:** No desenvolvimento de software, o planejamento e o produto final estão relacionados com quatro variáveis interdependentes: tempo, custo, escopo e qualidade. Essas variáveis se relacionam de forma que a alteração do valor de qualquer uma delas influencia as outras;

- **Priorização:** Métodos ágeis baseiam-se fortemente na adaptação a mudanças. As estratégias de planejamento focam em planos detalhados para o curto prazo e mais superficiais para o futuro distante. Desta forma, é possível uma visão panorâmica para guiar as decisões no longo prazo e precisão nas atividades do dia-a-dia. As duas principais vantagens desta abordagem são: 1) economia de esforço ao abrir mão do detalhamento de longo prazo, pois é difícil fazer previsões sobre o futuro e a alta chance de mudanças durante a execução invalida facilmente especificações minuciosas; 2) detalhes no longo prazo trazem a falsa sensação de certeza, pois os indivíduos passam a tratar previsões detalhadas como predições. Equipes ágeis revêem seus planos constantemente. A cada planejamento, elas têm a oportunidade de avaliar as condições do projeto e, baseadas nesses fatos, traçar o melhor caminho para atingir seus objetivos. Esta estratégia mantém os planos e a execução sempre adequados à realidade, que inevitavelmente estão em mutação, significando identificar prioridades para cada momento do projeto.

Diante da mudança de cultura que o desenvolvimento ágil causa nas empresas desenvolvedoras de *software*, algumas dificuldades podem surgir em sua implementação como: apoio das instâncias superiores, interação com outros departamentos e com os clientes.

5. Considerações Finais

O desenvolvimento de *software* tradicional não possui um histórico de bons resultados, como mostrado no "*Chaos Report*" [4], onde uma pesquisa realizada em projetos de TI mostrou que a grande parte dos projetos é entregue fora do prazo e ou do custo. E também se notou que o uso das funcionalidades disponibilizadas é baixíssimo. O resultado são projetos com grandes desperdícios e clientes insatisfeitos.

As metodologias ágeis são interativas e incrementais, resultando em um produto desenvolvido com base na melhoria contínua, e como o cliente participa de todo o projeto, a sua satisfação normalmente é garantida.

Este trabalho conclui as metodologias ágeis apresentadas neste poderiam ser aplicadas de maneiras complementar ou alternativa às metodologias tradicionais. Conclui-se também que o Scrum, XP e o *Lean Software Development* poderiam ser empregados de maneira complementar entre si. O Scrum é um *framework* focado principalmente em planejamento e gerência. Já o XP é mais focado em práticas de

desenvolvimento. Mesmo assim, várias práticas são coincidentes entre Scrum/XP como *sprint*/desenvolvimento iterativo, *daily scrum/daily meeting*, *sprint planning/planning game* e assim por diante. O *Lean Software Development* entra como um conjunto de princípios, valores e ferramentas para um desenvolvimento enxuto. A Figura 5 a seguir apresenta a diagramação proposta:
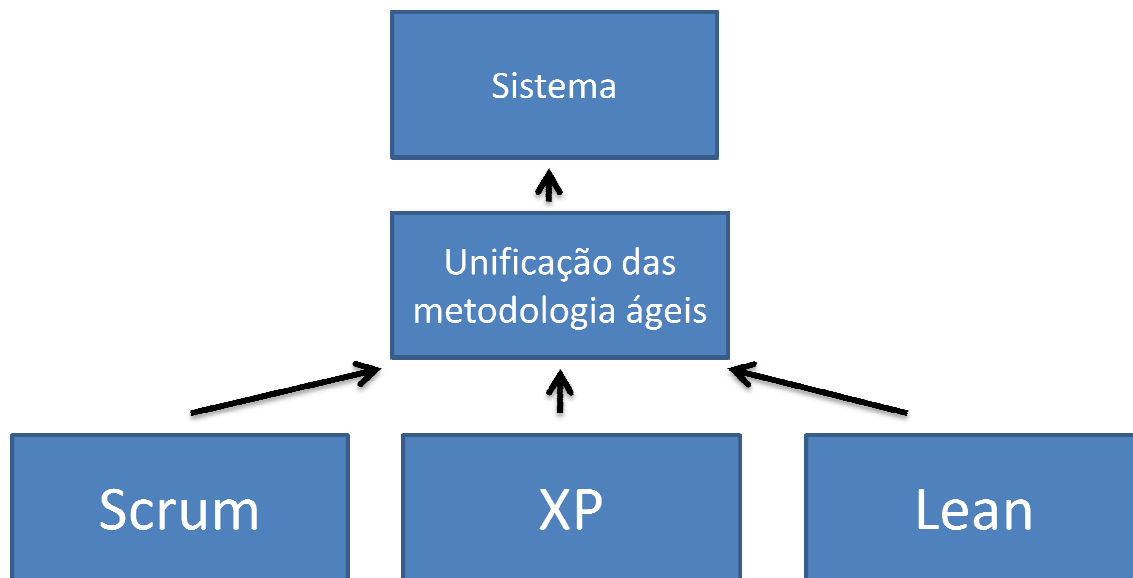
Figura 5. Complementaridade dos métodos ágeis.

6. Referências Bibliográficas

[1] ARAUJO, R. C., GALINA, T. C.: Análise de Escopo e Planejamento no Desenvolvimento de Software, sob a Perspectiva Ágil. Universidade do Vale do Rio dos Sinos (Unisinos). São Leopoldo – RS – Brasil. 2007.

[2] BECK, K.: Extreme Programming explained: Embrace change. Reading, Mass., Addison-Wesley, 244 p. 1999.

[3] BECK, K., et al.: Manifesto for Agile Software Development. 2001. Disponível em <http://www.agilemanifesto.org>. Acesso em Junho de 2010.

[4] CHAOS REPORT. Disponível em < www.standishgroup.com>. Acesso em Junho de 2010.

[5] DEMING, W. E.: Out of the Crisis: Quality, Productivity, and Competitive Position. Cambridge University Press .1986.

[6] DYBA, T., DINGSØYR, T.: What do we know about Agile Software Development? Published by the IEEE Computer Society. 2009.

[7] FILHO, D. L. B.: Experiências com desenvolvimento ágil. Instituto de Matemática e Estatística da Universidade de São Paulo (Dissertação de Mestrado). 2008.

[8] FRANCO, E. F.: Um modelo de gerenciamento de projetos baseado nas metodologias ágeis de desenvolvimento de software e nos princípios da produção enxuta. Escola Politécnica da Universidade de São Paulo (Dissertação de Mestrado). 2007.

[9] LEAN SOFTWARE INSTITUTE.: Introducing Lean Software Development. Disponível em <http://www.leansoftwareinstitute.com/art_ilsd.php>. Acesso em maio de 2010.

[10] KALERMO, J., RISSANEN, J.: Agile software development in theory and practice. Universidade de Jyväskylä, Finlândia (Dissertação de Mestrado). 2002.

[11] LEAN INSTITUTE BRASIL. Disponível vem <http://www.lean.org.br>. Acesso em Março de 2010.

[12] TOYOTA MOTOR MANUFACTURING. Disponível em <http://www.toyotageorgetown.com/history.asp >. Acesso em Março de 2010.

[13] PARNELL-KLABO, E.: Introducing Lean Principles with Agile Practices at a Fortune 500 Company. Proceedings of AGILE 2006 Conference. 2006.

[14] POPPENDIECK, M.: Lean Software Development. 29th International Conference on Software Engineering. IEEE. 2007.

[15] POPPENDIECK, M., POPPENDIECK, T.: Lean Software Development: An Agile Toolkit for Software Development Managers. Primeira Edição. Boston: Addison-Wesley Professional. 2003.

[16] SCHWABER, K.; BEEDLE, M. Agile Software Development With Scrum. Primeira Edição. Upper Saddle River: Prentice-Hall. 2001.